String Pattern Matching Algorithms

Raphael Ribeiro da Costa e Silva **Supervisor: Dr. Carlos E. Ferreira**

Institute of Mathematics and Statistics. University of Sao Paulo

Introduction

An **alphabet** Σ is a finite set. A **symbol** is an element $s \in \Sigma$. A **string** is a finite sequence of symbols, i.e, elements of an alphabet. The set of all strings over Σ is denoted by $\Sigma*$. We say a string $p \in \Sigma*$ is a **pattern** over a fixed alphabet Σ if p consists of symbols from Σ. Let s be a string and denote $s := s_1 \cdots s_n$. We say that a subsequence $s_i \cdots s_j$ of s if a **substring** of s. An **ocurrence** of u in s is a pair (i, j) such that $u := s_i \cdots s_j$ is a substring of s.

Let P be a set of patterns over a fixed alphabet Σ and let T be a fixed string called text input. The Multiple Pattern String Matching (MPSM) is the problem of finding all occurrences of all patterns of P in T . The Single Pattern String Matching (SPSM) is a special case of (MPSM) by adding the constraint $|P| = 1$.

The String pattern matching problem is one the most relevant string problems. There are practical solutions to real-world problems that can be developed using these algorithms, including, but not limited to, intrusion detection systems, evolutionary biology, computational linguistics, and data retrieval.

Boyer-Moore

The Boyer-Moore Algorithm is one of the most famous algorithms to solve the String Pattern Matching Problem. It uses the idea of the brute-force algorithm improving it by using some heuristics: Bad character rule and Good Suffix Rule. We align the pattern p with the text $\mathcal{T}$ and start checking the match character by character, in reverse order of p. If a mismatch occurs we have to shift the pattern and try again.

When a mismatch occurs, we have to shift the pattern until we no longer have a mismatch, i.e, until we found a match, or until the pattern move past the mismatched character. The **Bad Character Rule** tells us to shift the pattern until we align it with the rightmost occurrence of the mismatched character.

	0	1	2	3	4	5	6	7	8	9
T	a	b	b	a	d	a	b	a	c	b
P	a	b	c	a	b	c				
shift			a	b	c	a	b	c		

Table 1: In this example, a mismatch occurs when comparing a-c. Then, we shift the pattern so that the letter a it is aligned with the rightmost occurence of a in P.

The second rule we are going to apply is the **Good Suffix Rule** and it is applied regarding the borders of the pattern. A **border** is a substring of p that is both a proper suffix and a proper prefix of p. Let t be the substring of $\mathcal{T}$ which is matched to the pattern at some iteration i and we have found a mismatch. We can safely shift the pattern until t is aligned with the rightmost ocurrence of $t \in \mathcal{P}$. See this example:

	0	1	2	3	4	5	6	7	8	9	10	11
T	a	b	c	a	b	c	**a**	**d**	**a**	a	b	c
P	b	a	d	a	a	a	**a**	**d**	**a**			
shift						b	a	d	a	a	a	a

Table 2: A mismatch occurs comparing c-a. In this case, t = ada. We can shift the pattern until it is aligned with the rightmost occurence of ada, which occurs at index 1.

The best case occurs when at each attempt the text character compared does not occur in the pattern and the pattern is shifted. In such case the algorithm runs in $\mathcal{O}(|\mathcal{T}|/|\mathcal{P}|)$. However, in general $O(|P||T|)$ comparisons are needed.

Shift-Or Algorithm

Let $p := p_0 \cdots p_n$ be a pattern of size n. The Shift-Or Algorithm constructs a hash table mask that maps each character c of p to a bitmask $d = d_n \cdots d_1$, where $d_i = \{0, 1\}$. mask[c] has I-th bit set to 0 if, and only if, $p_i = c$. For instance, for the pattern $abra$, we have the following table:

character	d_1	d_2	d_3	d_4	bitmask $(d_4 \cdots d_1)$
a	0	1	1	0	0110
b	1	0	1	1	1101
r	1	1	0	1	1011

Table 3: In this case, a mismatch occurs comparing c-b. So we shift the pattern to align it with the next rightmost occurrence of c in P

With the table mask created, we can find the occurrences of the pattern p in the text T with the following algorithm:

1. We start with a bitmask ϕ with all bits set to 1.
2. For each character c of $\mathcal{T}$, we perform the OR bitwise operator with ϕ and $mask[c]$ and shift phi to the left. Thus, we apply the assignment:

$$(\phi | mask[c]) << 1$$

3. If at any step, the d_m bit is set to 1, then a match was found at index $i - n + 1$.

The complexity of the Shift-Or Algorithm is $\mathcal{O}(|\mathcal{T}| + |p|)$. Since it mostly uses bitwise operations to perform the tasks.

Trie of Suffixes

A trie is a useful data structure to represent set of strings. Suppose we have a set of strings $\mathcal{S}$. For each each string $s \in \mathcal{S}$ we add its character, node by node, in a Tree structure.

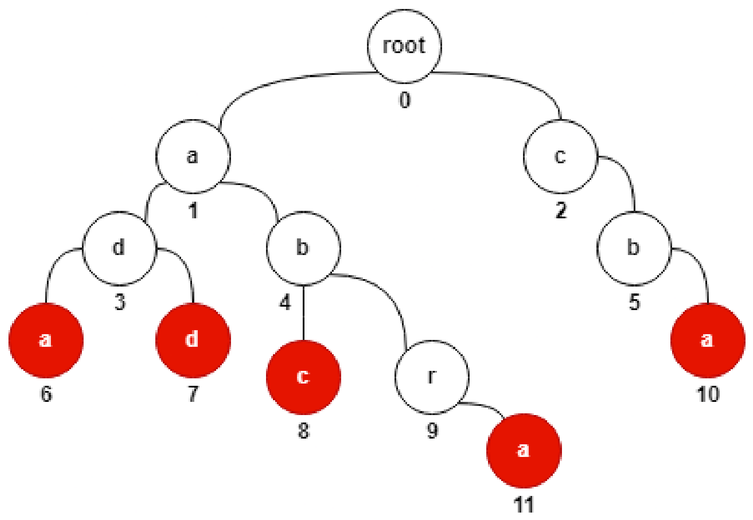

Figure 1: Trie Data Structure

We can use this data structure to solve the Single pattern string matching problem by constructing a trie with all suffixes of $\mathcal{T}$.

Then, we can process each character of p, starting from the root, and following the edges of the trie for each character. If, at any moment, there is no such edge, then p doesn't occur in $\mathcal{T}$. If we reach a leaf node, then we have found a match.

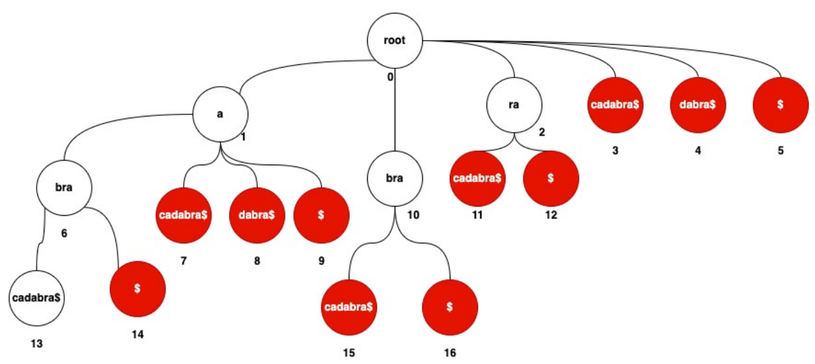

Figure 2: Trie Data Structure

The preprocessing phase is $\mathcal{O}(|\mathcal{T}|^2)$ of complexity and the search phase is $\mathcal{O}(|\mathcal{T}| + |\sigma|)$ where σ is the set of occurences of P in $\mathcal{T}$. However, there are advanced algorithms to improve the preprocessing phase of this algorithm.

Aho-Corasick

The Aho-Corasick algorithm is one of the algorithms to solve the Multiple Pattern String Matching Problem. We are going to describe how to construct a **finite deterministic automaton** from the Trie Data Structure. Suppose we have a set of pattern $\mathcal{P}$ and a Trie $\mathcal{T}$ for the set of strings $\mathcal{P}$. We construct an automaton in which every vertice v of $\mathcal{T}$ is a state, and for every edge e of the trie, we have a transition according to the corresponding letter. Notice that this doesn't define an automaton yet, since we have to define a transition from every letter from the alphabet. If there is no corresponding edge in the trie, then we have go into some state. Which will be defined by using the suffixes links.

A **suffix link** is an edge that leads to the proper suffix of the string $s[v]$. We build such links recursively. The base case is the root of the trie, in which the suffix link will point to itself. The general case we have some vertex v, and there is no transition from v, with a letter c. Then, we can go to the ancestor p of v, follow its suffix link, which is already defined by induction, leading to some vertex u, and try to perform the transition with the letter c from u. If there is no such edge, we repeat until we reach the root, our base case. Therefore, we can build such links in linear time proportional to the size of the Trie. We also create exit links, which points to the nearest leaf vertex that is reachable using suffix links, this will be used to find all matches from a leaf node. We can construct such links lazily in linear time.

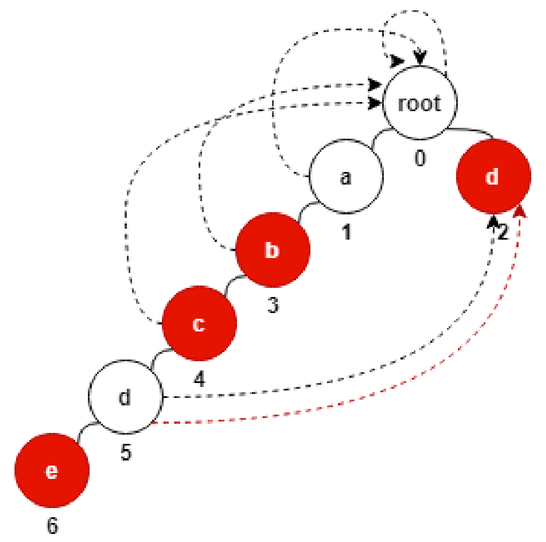

Figure 3: Aho Corasick Data Structure, now with exit links added, for $\mathcal{T} = abcd$ and $\mathcal{P} = [ab, abc, abcde, d]$.

After building the automaton, we process each character c in $\mathcal{T}$, starting from the root, and following the edges. If there is no edge for the current character, we follow the suffix link and try again. If we reach a leaf node, we have found a match and we can find all other matches by following the exit links.

The complexity of the Aho-Corasick Algorithm is $\mathcal{O}(|\mathcal{T}| + |\sigma|)$ where σ is the set of occurences of P in $\mathcal{T}$

Monography

Monography: https://linux.ime.usp.br/~raphaelrbr/mac0499

Email: raphaelrbr@usp.br